REMARKS

Applicants respectfully request reconsideration in view of the above amendments and the following remarks. Applicants add claims 22-26, entry is requested. Applicants do not amend or cancel any claims. Accordingly, claims 1-26 remain pending in the application.

I. Amendments to the Claims

Applicants add claims 22-26, which depend from claims 1, 6, 12, 15 and 19, respectively. Support for the new dependent claims may be found in the specification at ¶¶ [0025] and [0029]. The amendments to the claims are therefore supported by the specification and do not add new matter. For at least the foregoing reasons, Applicants respectfully request consideration and entry of the attached amendments to the claims.

II. Claims Rejected Under 35 U.S.C. § 102

Claims 6-21 stand rejected under 35 USC section §102(e) as being anticipated by U.S. Patent No. 6,807,621 issued to Strombergson et al. (hereinafter "Strombergson").

To anticipate a claim, a single reference must disclose each element of that claim. In regard to claim 6, this claim includes the elements of "tracking program order of the first set of instructions relative to the second set of instructions in a global reorder buffer." Applicants believe that Strombergson does not teach these elements of claim 6. Examiner cites commit stage 5 in Fig. 1 of Strombergson, and asserts that "since the commit stage contains a reorder buffer (ROB 10), the stage is responsible . . . for tracking program order from all the execution stages." Examiner further asserts that "commit stage 5 . . . must obtain each of the instructions previously contained in the local buffers. The contents of these local buffers must be merged

precisely back into the program order for the processor to work appropriately." Examiner concludes that therefore, "[t]he relative order must be tracked; otherwise, these instructions could not possibly be merged back into the original program order."

Applicants assume that Examiner is asserting that the elements of claim 6 are inherently taught by Strombergson, since Examiner has not relied on Strombergson for explicit disclosure of "tracking program order of the first set of instructions relative to the second set of instructions in a global reorder buffer." As noted previously on page 11 of the Amendment and Response to Office Action filed by Applicants on February 24, 2006, Examiner's assertions rely on information extrinsic to the cited reference and are not proper bases for an anticipation rejection.

Further, it is not inherent in Strombergson that the *relative* order is tracked *by commit stage 5*. For example, assume that the decoding stage 2 (Strombergson, Fig. 1) places an entry for the instruction in reorder buffer 10 of commit stage 5. *See* Patterson & Hennessy, *Computer Organization & Design* 517 (2d ed. 1998) (submitted herewith). In such a case, although commit stage 5 is keeping track of the overall program order of instructions, it is simply receiving the instructions already in program order, from decoding stage 2 to reorder buffer 10. Thus, in this example, the commit stage is keeping track of the overall program order (or *absolute* order) via the reorder buffer, while the commit stage does not track program order of a first set of instructions *relative* to a second set of instructions, to properly maintain program order of the instructions.

Further, a standard reorder buffer does not track the relative order of instructions that have been assigned to different execution units. Rather, a standard reorder buffer is a queue in which the entire sequence of the instructions is placed that are being operated on by all execution units. As the execution units complete their execution of instructions the results are placed in the appropriate slot of the queue in the standard reorder buffer. *See High Performance Micro*

Processors: Chapter 8 at www.cs.swan.ac.uk/~csneal/hpm/reorder.html (last visited Feb. 24, 2006) (and submitted herewith). The Examiner has not indicated and the Applicants have been unable to discern any part of Strombergson that teaches that the reorder buffer in the commit stage is anything other than a standard reorder buffer.

Therefore, Strombergson does not teach each of the elements of claim 6. Accordingly, reconsideration and withdrawal of the anticipation rejection of claim 6 are requested.

Claims 7-11 depend from independent claim 6 and incorporate the limitations thereof. Thus, at least for the reasons mentioned above in regard to independent claim 6, these claims are not anticipated by Strombergson. Accordingly, reconsideration and withdrawal of the anticipation rejection of these claims are requested.

In regard to independent claims 12, 15 and 19, these claims include elements similar to those of independent claim 6 including "a global reorder buffer to track instruction order of instructions assigned to the first reorder buffer relative to the second reorder buffer," "tracking program order of the first set of instructions relative to the second set of instructions in a global tracking device" and "a means for directing program order of the first set of instructions relative to the second of instructions and the global tracking device." Thus, at least for the reasons mentioned above in regard to independent claim 6, Strombergson does not anticipate each of these claims. Accordingly, reconsideration and withdrawal of the anticipation rejection of these claims are requested.

Claims 13, 14, 16-18, 20 and 21 depend from independent claims 12, 15 and 19, respectively, and incorporate the limitations thereof. Thus, at least for the reasons mentioned above in regard to independent claims 12, 15 and 19, these claims are not anticipated by Strombergson. Accordingly, reconsideration and withdrawal of the anticipation rejection of these claims are requested.

Claims 1-5 stand rejected under 35 USC section §102(e) as being anticipated by Strombergson.

In regard to claim 1, this claim includes elements similar to those of independent claim 6 including "a third device coupled to the first device and second device to track relative segment order between the first device and the second device." Thus, at least for the reasons mentioned above in regard to independent claim 6, Strombergson does not anticipate claim 1. Accordingly, reconsideration and withdrawal of the anticipation rejection of this claim are requested.

Claims 2-5 depend from independent claim 1 and incorporate the limitations thereof. Thus, at least for the reasons mentioned above in regard to independent claim 1, these claims are not anticipated by Strombergson. Accordingly, reconsideration and withdrawal of the anticipation rejection of these claims are requested.

CONCLUSION

In view of the foregoing, it is believed that all claims now pending, namely claims 1-26, patentably define the subject invention over the prior art of record, and are in condition for allowance and such action is earnestly solicited at the earliest possible date. If Examiner believes that a telephone conference would be useful in moving the application forward to allowance, Examiner is encouraged to contact the undersigned at (310) 207-3800.

Respectfully submitted,

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP

Dated: October 31, 2006

Jonathan S. Miller Reg. No. 48,534

12400 Wilshire Boulevard, Seventh Floor
Los Angeles, California 90025
(310) 207-3800

CERTIFICATE OF FACSIMILE TRANSMISSION:

I hereby certify that this correspondence is being transmitted via facsimile to 571-273-8300 addressed to: Mail Stop AF, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

Annie McNally 10/31/2006
Annie McNally Date

Chapter 8: Reorder Buffers and Register Renaming

We have seen a range of possible ways that execution in a pipelined/superscalar machine could be delayed, but we have looked at actual mechanisms for reducing the effects of only some of them. For example, various strategies for *procedural dependency* (e.g. branch prediction, etc.), and have mentioned (if briefely) a strategy for resource conflict (duplication of resources). We have done nothing for those conflicts that involve data - i.e. true data dependency, output dependency and antidependency. We have not even made it clear, except informally, how we would go about recognizing their presence. In this chapter and the next we will start to address this. In particular, in this chapter, we will look at part of the solution to eliminating WAW and WAR hazards arising from output and antidependency, and mechanisms to enforce a precise architectural state. It turns out that the same hardware is involved in both of these steps.

As we saw earlier true data dependency is a property of a program, but name dependencies are not and can potentially be eliminated. Usually name dependencies involve conflicts in register use. One possible approach is to provide as many registers as possible, in an attempt to avoid clashes by giving the compiler plenty of choice. (This is basically the same solution used for resource conflicts - provide more resources.)

One problem is backward compatability with existing architectures - these often have a limited number of registers, and there is no way to increase this number without radically redefining the architecture. Another problem is that a large register file means more work saving/restoring it when changing between processes (*context switching*).

8.1. Register Renaming

A devious strategy is *register renaming* - the hardware has a larg(ish) set of registers - often several times as many as the actual architecture claims to have. These registers are not associated permanently with the registers of the architecture, but are *dynamically* allocated as needed. Furthermore, there can be several versions of an architectural register present at any one time. Consider the following code:

```
MUL R2,R2,R3    ; R2 = R2 * R3
ADD R4,R2,1     ; R4 = R2 + 1
ADD R2,R3,1     ; R2 = R3 + 1
DIV R5,R2,R4    ; R5 = R4 * R4
```

Consider one problem: instruction 3 cannot go ahead until instruction 1 has finished, and instruction 2 has started. This is because there is an output dependency between instruction's 1 and 3 - both write to R2 - and an antidependency between instructions's 2 and 3 - instruction 3 overwrites instruction 2's argument. Now consider the same program, but with the registers labelled.

```
MUL R2_b,R2_a,R3_a
ADD R4_b,R2_b,1
```

```
ADD R2_c,R3_a,1
DIV R5_b,R2_c,R4_b
```

Now instruction 3 can start immediately, because it is using a 'different' R2 from instructions 1 and 2. We are effectively using a *history* of the contents of each register - for example, `R2_c` is the newest version of `R2`, then `R2_b`, followed by `R2_a` (the oldest version).

There are two ways we can go about implementing register renaming. The first is by explicitly providing a larger set of registers than the architecture claims is present - a technique usually simply called register renaming. Alternatively (and more usually) by using a *reorder buffer*. We will look at reorder buffers first, and then briefly consider the alternative.

8.2. Reorder Buffers

Renaming based on a reorder buffer uses a physical register file that is the same size as the architectural register file, together with a set of registers arranged as a *queue* data structure, called the *reorder buffer*.

As instructions are issued, they are assigned entries for any results they may generate at the tail of the reorder buffer. That is, a place is reserved in the queue. We maintain the logical order of instructions within this buffer - so if we can issue four instructions `i` to `i+3` at once, we put `i` in the reorder buffer first, followed by `i+1`, `i+2` and `i+3`. As instruction execution proceeds, the assigned entry will ultimately be filled in by a value, representing the result of the instruction. When entries reach the head of the reorder buffer, provided they've been filled in with their actual intended result, they are removed, and each value is written to its intended architectural register. If the value is not yet available, then we must wait until it is. Because instructions take variable times to execute, and because they may be executed out of program order, we may well find that the reorder buffer entry at the head of the queue is still waiting to be filled, while later entries are ready. In this case, all entries behind the unfilled slot must stay in the reorder buffer until the head instruction completes. For example, consider the case of 6 instructions I1 - I6. Suppose at a given clock cycle that I1 and I2 both finish, and that at earlier clock cycles I4 to I6 also finished, but I3 is yet to complete. We can move the results for I1 and I2 out of the reorder buffer into their respective architectural registers. However, I4 to I6 must wait until I3 has completed. Fig 1 illustrates the basic idea.

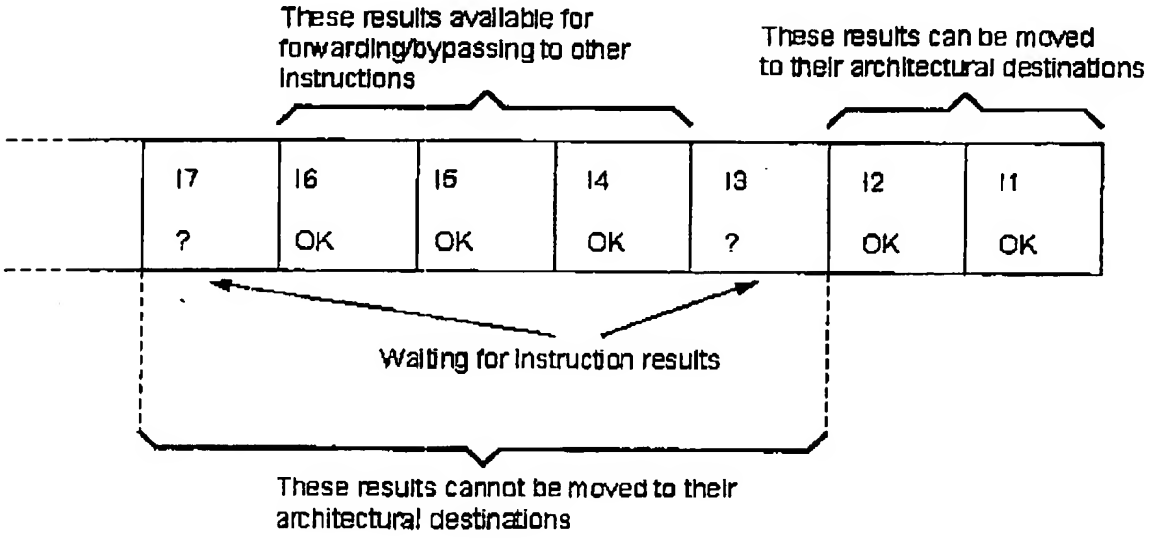

Fig.1. A Reorder Buffer

As described, the reorder buffer does not solve our problems - though is does solve another one (see below). However, even though the results of some instructions (I5 and I6 above) cannot be moved to their architectural destinations, they can still be used in computations. Suppose, in the example above that we execute a further instruction I7, which uses some register R2 as an operand. Suppose further that the result of I5 will eventually be stored in R2, and this is the actual value required by I7. Even though the value in the reorder buffer computed by I5 has not yet been moved to R2, we can still use it in the computation of I7. This process is called *forwarding* or *bypassing*, and we have mentioned it already when we considered basic pipelining - though not seen how to implement it. The reorder buffer effectively provides the history mechanism required for register renaming. The oldest version of a register is that stored in the architectural register; the next oldest is that nearest the head of the reorder buffer; the youngest is that nearest the tail of the reorder buffer.

(In practice of course, the details of implementation are not that straightforward. Reorder buffer entries need to store considerable amounts of information about instruction results - the instruction, its eventual destination, whether the result is valid or not - and it is important to be able to access all this information quickly. In order to avoid stalling the pipeline, we must be able to quickly identify when a result - needed by another instruction - becomes available, and also fetch it quickly.)

8.2.1. Maintaining a Precise Architectural States

The other problem that reorder buffers solve is that of maintaining a *precise architectural state*. Recall from an earlier chapter the problem of an instruction `i+1` terminating before instruction `i`, and then `i` causing an error. Reorder buffers solve this by effectively keeping instruction results provisional until earlier ones are known - provided instructions are actually issued in program order. Suppose in the example above that instruction I3 caused an error. We simply discard the contents of the reorder buffer (including the already executed instructions I4 to I6) and restart execution at I3 (if possible). We may waste some work (redoing I4 to I6), but we maintain backward compatibility, and a precise architectural state (because only the results of I1 and I2 will have been written out to architectural registers). Note that this is only actually the case if we *issue* the instructions in program order, and hence the order of the entries in the reorder buffer reflect program order. However, recall that we generally do issue instructions in order.

A final point to note is that reorder buffers do not cause any additional workload when we do a context switch, since the contents of the reorder buffer do *not* have to be saved. We do of course have to wait for any outstanding instructions to leave the buffer (or just dump the results and repeated the dumped instructions), but this is much quicker than the (slow) process of saving registers to memory.

8.3. Another Register Renaming Strategy

Reorder buffers are convenient and simple (at least conceptually). They are also widely used (for example, all P6-based processors (Pentium Pro, Pentium II and all Pentium III-series processors use reorder buffers). However, they are not completely without disadvantages. For example, they add an extra step to the pipeline - moving results from reorder buffer to architectural registers. What is more, there is generally a limit on the number of entries that can be moved simultaneously. For example, suppose there are six execution units. In principle, this means six instructions can complete simultaneously. In practice, this will not happen often - it is unlikely we will feel it is worthwhile designing a reorder buffer and register set that will allow six results to be moved at once. So when it does happen, or whenever more instructions complete than we can move simultaneously, the pipeline

will stall. (Actually it may not all stall - only the execution unit(s) that wait.) Also, a reorder buffer is an additional place that execution units (or issue units) must look for operands in addition to the actual registers, and generally somewhere else as well (as we will see when we look at algorithms). This again means more work to do, and potentially worse performance.

An alternative is instead to provide a large set of registers and *dynamically* decide at any point in time which ones represent the actual architectural registers. For example, in a machine with 16 architectural registers we might provide a set of 64 physical registers. At any point in time, only 16 of these will correspond with the actual architectural register set - precisely *which* 16 changing as the program executes. This method - usually just called Register Renaming - solves the problems above. However, it has problems of its own. One is deciding at any one time which registers are *live* - that is, contain results that are still needed. Note that the live registers are *not* exactly the same set of registers as those corresponding with the architectural ones - determining those is another problem. Most of the time it does not matter which registers correspond with the architectural ones. However, when a context switch occurs it is necessary to know which ones to save - so there must be a means of finding this out. This is potentially a fair amount of work - however much of it can be done in parallel with the operation of the rest of the pipeline, meaning it is not on the critical path and will not slow it down. The Pentium 4 has chosen to change from a reorder buffer to register renaming.

Previous Contents Next

Elaboration: Memory accesses benefit from *nonblocking caches*, which continue servicing cache accesses during a cache miss (see Chapter 7). Out-of-order execution processors need nonblocking caches to allow instructions to execute during a miss.

6.9 Real Stuff: PowerPC 604 and Pentium Pro Pipelines

Dynamically scheduled pipelines are used in both the PowerPC 604 and the Pentium Pro. They have such similar pipeline organizations that we use a single generic drawing, Figure 6.62, to describe both. Figure 1.18 on page 27 shows the silicon area required by dynamic pipelining in the Pentium Pro.

The instruction cache fetches 16 bytes of instructions and sends them to an instruction queue: four instructions for the PowerPC and a variable number of instructions for the Pentium Pro. Next, several instructions are fetched and decoded. Both processors use a 512-entry branch history table to predict branches and speculatively execute instructions after the predicted branch. The dispatcher unit sends each instruction and its operands to the reservation station of one of the six functional units. The dispatcher also places an entry for the instruction in the reorder buffer of the commit unit. Thus an instruction cannot issue unless there is space available in both an appropriate reservation station and in the reorder buffer.

With so many instructions executing at the same time, we can run out of places to keep results. Both processors have extra internal registers, called *rename buffers* or *rename registers*, that are used to hold results while waiting for the commit unit to commit the result to one of the real registers. The decode unit is where rename buffers get assigned, thereby reducing hazards on register numbers. Whenever an entry in the reservation station has all its operands and the associated functional unit is available, the operation is performed.

The commit unit keeps track of all the pending instructions in its reorder buffer. Because both machines use branch prediction, an instruction isn't finished until the commit unit says it is. When the branch functional unit determines whether or not a branch was taken, it informs both the branch prediction unit, so that it can update its state machine, and the commit unit, so that it can decide the fate of pending instructions. If the prediction was accurate, the results of the instructions after the branch are marked valid and thus can be placed in the programmer-visible registers and memory. If a misprediction occurred, then all the instructions after the branch are marked invalid and discarded from the reservation stations and reorder buffer.

The commit unit can commit several instructions per clock cycle. To provide precise behavior during exceptions, the commit unit makes sure the instruc-

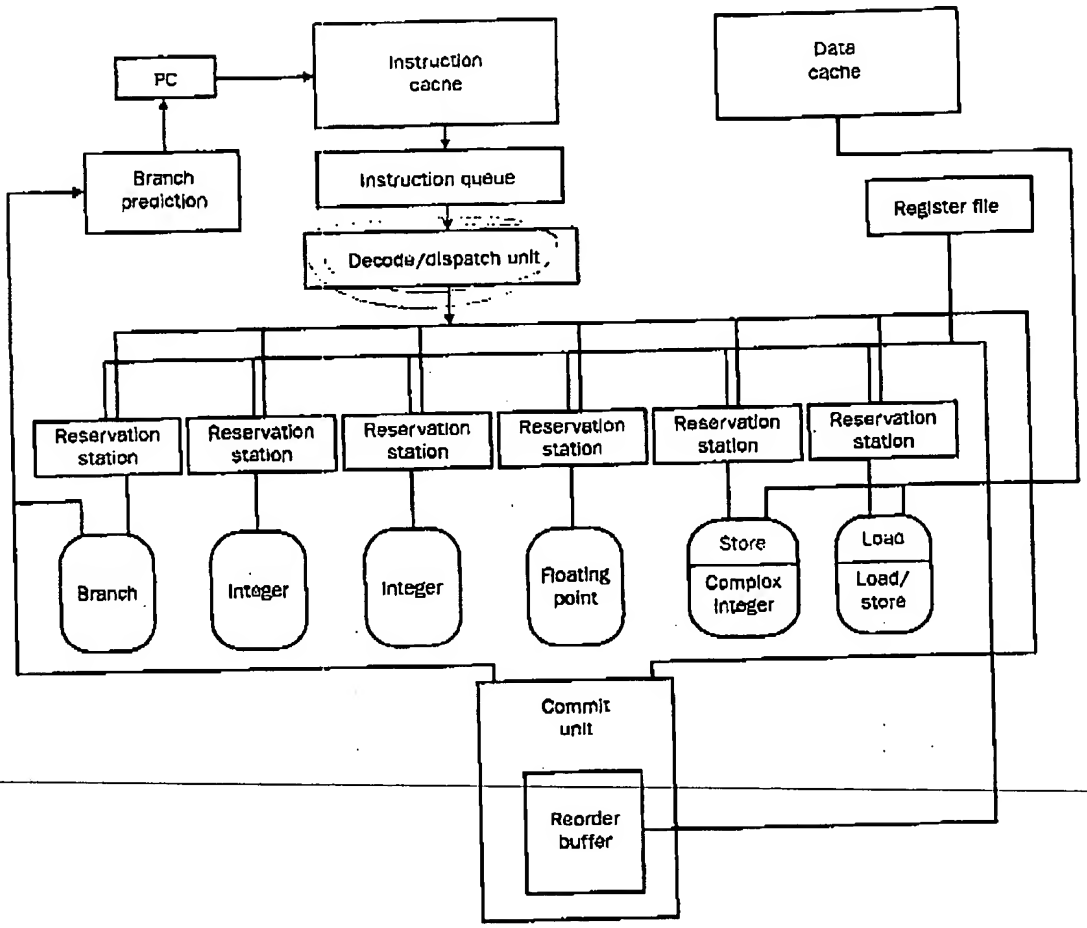

FIGURE 6.62 The generic pipeline organization of the Intel Pentium Pro and the PowerPC 604. Both have six functional units, and the first four have the same responsibilities. For the Pentium Pro, the last two functional units are Store and Load (in color), and the PowerPC 604 has Complex Integer and Load/Store functional units. The figure shows that every functional unit has its own path to a reservation station, but the Pentium Pro has a single central reservation station that can be used for any functional units with one bus shared by the branch and one of the integer units, one bus shared by the other integer unit and floating-point unit, and separate buses for the rest of the functional units. The names for the major units on the Pentium Pro are Fetch/Decode Unit, Dispatch/Execute Unit, and Retire Unit, and the functional units are called Jump Execution Unit (EU), Integer EU, Floating Point EU, Store Address Generation Unit (AGU), and Load AGU. The real names for the major units on the PowerPC are Instruction Unit, Functional Units, and Completion Unit, and the functional units are called Branch Processing Unit, Single-Cycle Integer Unit, Floating Point Unit, Multi-Cycle Integer Unit, and Load/Store Unit.

tions commit in the order they were issued. Thus the commit unit cannot commit an instruction until the operation is finished in the functional unit, all branches on which it might depend are resolved, and all instructions issued before it have committed.

Figure 6.63 lists the specific parameters for the PowerPC 604 and Pentium Pro pipelines. Many of the differences between the Pentium Pro and the PowerPC 604 are cosmetic. The largest difference, not surprisingly, is in decode and dispatch. As described in section 5.7, rather than try to pipeline variable-length 80x86 instructions, the Pentium Pro decode unit translates the Intel instructions into 72-bit, fixed-length microoperations, and then sends these microoperations to the reorder buffer and reservation stations. This translation takes 1 clock cycle to determine the length of the 80x86 instructions and then 2 more to create the microoperations.

As discussed in Chapter 5, these microoperations have two source registers and one destination register, and are similar to MIPS instructions. Most 80x86 instructions are translated into one to four microoperations, but the really complex 80x86 instructions are executed by a conventional microprogram that issues long sequences of microoperations.

Parameter name	PowerPC 604	Pentium Pro
Maximum number of instructions issued per clock cycle	4	3
Maximum number of instructions completing execution per clock cycle	6	5
Maximum number of instructions committed per clock cycle	6	3
Number of bytes fetched from instruction cache	16	16
Number of bytes in instruction queue	32	32
Number of instructions in reorder buffer	16	40
Number of entries in branch table buffer	512	512
Number of history bits per entry in branch history buffer	2	4
Number of rename buffers	12 integer + 8 FP	40
Total number of reservation stations	12	20
Total number of functional units	6	6
Number of integer functional units	2	2
Number of complex integer operation functional units	1	0
Number of floating-point functional units	1	1
Number of branch functional units	1	1
Number of memory functional units	1 for both load and store	1 for load + 1 for store

FIGURE 6.63 Specific parameters of the PowerPC 604 and Pentium Pro in Figure 6.62.